Advanced Databases

Lectures
October 2014.

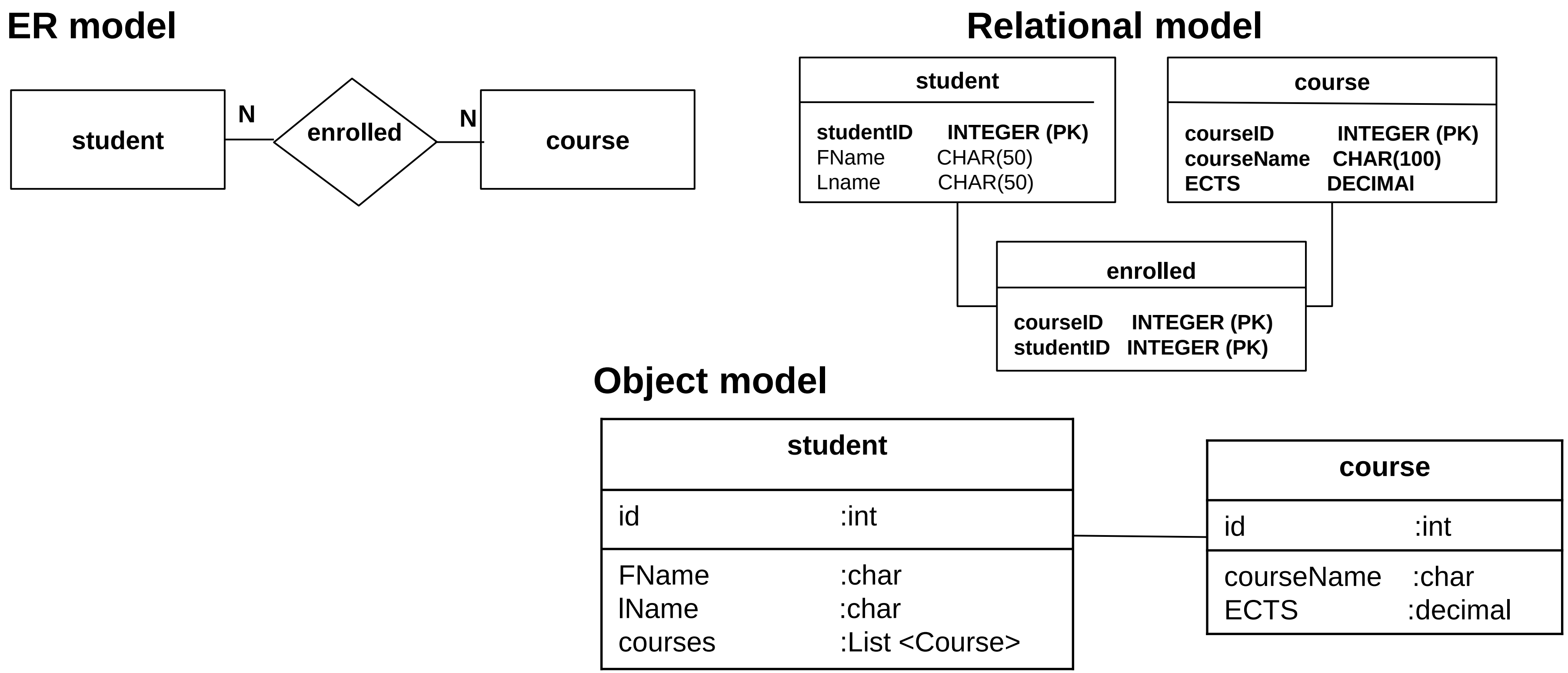
3. Object-oriented and object-relational databases

Overview

- Object-oriented database
  - The principles of object-oriented database
  - Object-oriented database management systems
  - ODMG standard
- Object-relational database
  - Object-relational data model
  - Object-relational features of SQL Standard

Object-oriented databases - motive (1)

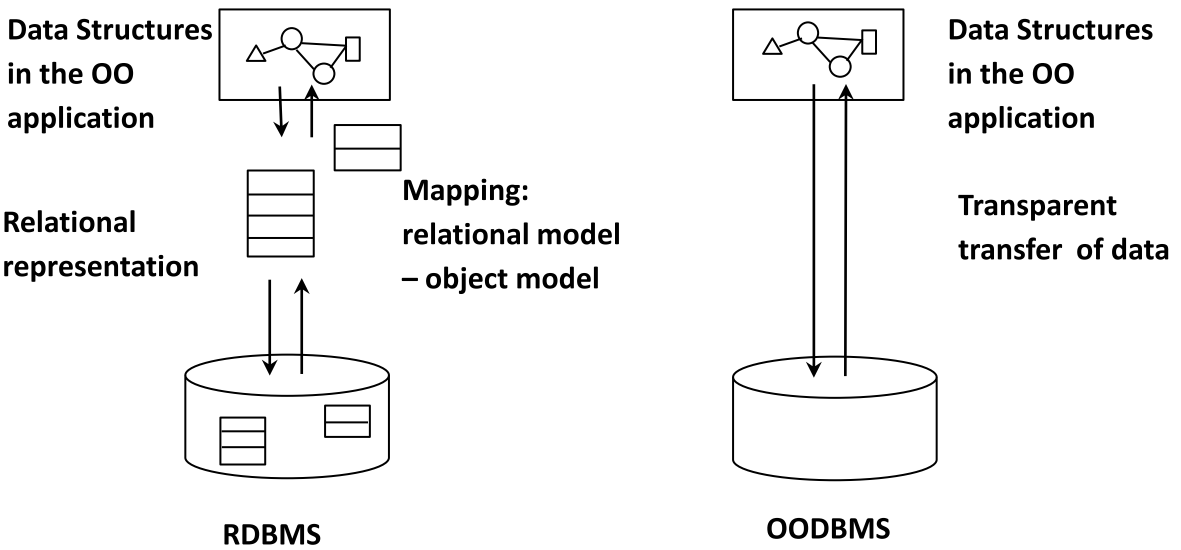
- Relational databases are not suitable for applications that use complex data types or new data types for large unstructured objects (unstructured text, images, multimedia, GIS objects, ...)
- Relational database model is very different from the object model implemented in object-oriented languages (Java, C #)



Object relational impedance mismatch

Object-oriented databases - motive (2)

- Object relational impedance mismatch
- Mapping between the two models is a tedious job – there is a need for transparent handling of data from the relational database, using the paradigm of object-oriented languages



Object-oriented databases - motive (3)

- **Object relational impedance mismatch:**
  - **Relationships between entities**:

    Relational model: relations student, studentCourse, course
    - using primary and foreign keys

    Object model: `student.getCourses()`
    - References to other objects
  - **Querying data**:

    Relational model:
    ```
    SELECT course.courseName
      FROM student, studentCourse, course
     WHERE ......
    ```
    - SQL (DDL, DML)

    Object model: OQL (Object Query Language), SODA (Simple Object Data Access), using object graf (`student.getCourses().get(0).getCourseName()`)
  - **Inheritance is not supported in the relational model**

Object model – relational model

Object-oriented model	Relational model
Class	Relational schema
Object	Entity, tuple
Member variable	Attribute
Method	Procedure
-	Primary key
OID	-

What element(s) from object model suits to relation from relational model?

Object-oriented database

- Object-oriented databases are sometimes called *object databases*
  - In the database are stored objects - the database model does not differ from the one in the application
  - Implementation of object-oriented database management system (OODBMS) is generally programmed for a specific programming language, and differ quite with each other
- *OODBMS is a database management system that implements object-oriented data model*
- *The Object-oriented Database System Manifesto, Atkinson at all,* 1989. – in scientific paper described the properties which OODBMS must satisfy
  - Object-oriented concepts
  - Database management system concepts

The basic principles of the object-oriented database/database management system

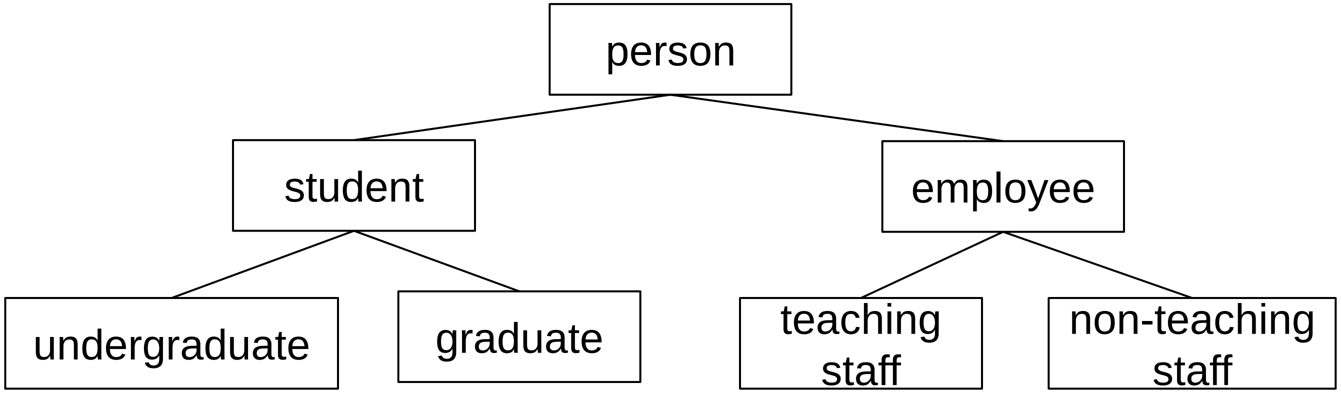
- Object-oriented concepts
  - Classes
  - Complex objects
  - Object identity
  - Class hierarchy
  - Encapsulation
  - Overriding, overloading and late binding
- Database management system concepts
  - Data persistence
  - Physical organisation of the data (*secondary storage management*)
  - Concurrency control
  - Database recovery
  - Ad Hoc Query Facility

Object identity - OID

- Unique, unchangeable object identifier generated by the OO system
- Independent of the values of the object attributes
- Invisible to the user
- Used for referencing objects
- Two objects are identical if they have the same identity of the object - property that uniquely identifies them

- In relational databases
  - the identity of the entity is based on the data values
  - primary key is used to ensure uniqueness
    - primary keys do not provide the kind of unity that is required for the OO systems:
      - keys are unique in the relation, not in the entire database
      - keys are mainly based on the attributes of the relation, which makes them dependent on the state of the object

Class hierarchy




```
public abstract class Person {
    private String personID;
    private String FName;
    private String LName;
     ...
    // access methods and constructors
     ...
}
```

```
public class Student extends Person {
    private String studentIDNumber;
     ...
    // access methods and constructors
     ...
}
public class Employee extends Person {
    private float salary;
     ...
    // access methods and constructors
     ...
}
...
```

- Available attributes and methods are inherited from the parent class
- **Subclasses can define new attributes and methods**
- **Generalization (*person*) and specialization (*undergraduate student*)**

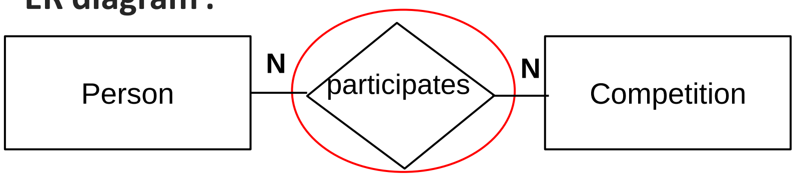
Relationships between objects

- Achieved with **referencing:**
- 1:1 relationship
  - `customer.getShoppingCart()`
  - `ShoppingCart.getCustomer()`
- N:1 (1:N) relationship
  - `customer.getResidenceCity()`
  - `city.getCustomers()`
- N:N relationship
  - `customer.getArticles()`
  - `article.getCustomers()`
- All connections can be two-way

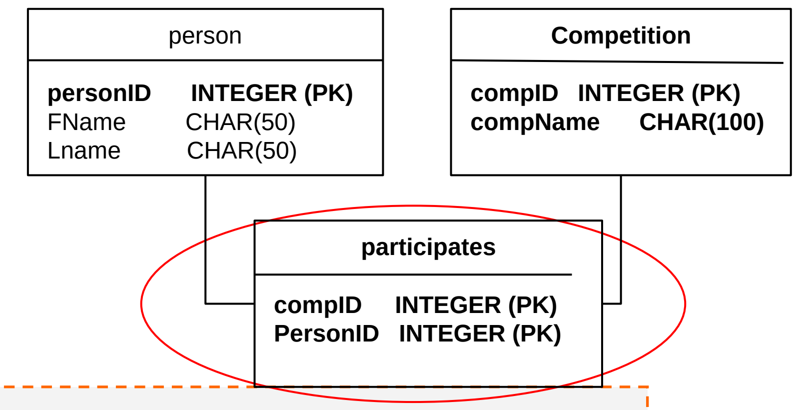
- Mutual references (two way) is not necessary to express in the object model, if it is not important for business process applications

Relationships between objects (2)

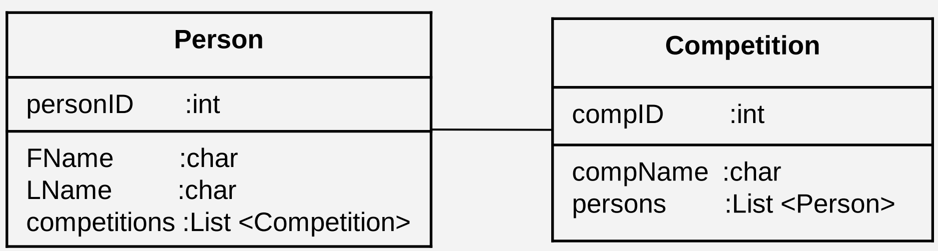
ER diagram :



relational diagram :



Object model



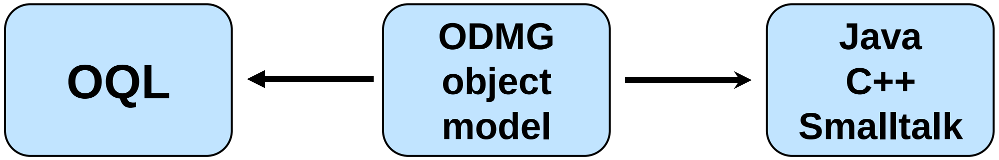
Class definition:

```
public class Person {
    private String personID;
    private String FName;
    private String LName;
    private List<Competition> competitions;
    …
    // access methods and constructors
    …
}
```

```
public class Competition {
    private int compID;
    private String compName;
    private List<Person> persons;
    …
    // access methods and constructors
    …
}
```

ODMG standard

- ODMG standard consists of the following:
  - Object model (OM)
  - Language for specifying objects (ODL - Object Definition Language)
  - Object query language (OQL)
  - Binding between the ODL and the object programming languages
    - Includes ODL which is dependent on the selected programming language
    - Provides an application programming interface (API) for the mapping of data types



OQL - Object Query Language (1)

- Query language for object databases modelled on SQL
- Flexible, but because of its complexity no manufacturer has fully implemented it
- Differences between OQL i SQL:
  - OQL supports referencing to objects within the table. Objects can be nested in other objects.
  - OQL does not support all of the keywords from the SQL
  - OQL supports mathematical calculations within OQL expression
- Syntax:
  - Queries in form using *Select-From-Where*

  or

  - Navigation in complex structures :

```
book.publisher.contact.email
```

OQL - Object Query Language (2)

The result of the OQL query is an object whose type depends on the operands involved in the query.

Example 1: Get the names and price of food on offer in the restaurant "Snack":

```
SELECT s.dish.name, s.price
     FROM Sells s
     WHERE s.restaurant.name = "Snack"
```

Example 2: Get the names and price of food, using object *restaurant:*

```
     SELECT s.dish.name, s.price
        FROM Restaurants r, r.dishesSold s
      WHERE r.name = "Snack"
```

OODBMS – advantages

- Better and faster to manage complex objects and relationships in comparison to relational systems
- Support hierarchy, class and inheritance
- Single data model
- Objects in the database and the objects in the application are the same, so there is no need for the two models
  - No object-relational mismatch
- There is no need for a primary key (?)
  - The identification of objects is hidden from the user
- Used only one programming language (application and database access)
- There is no need for a special query language

OODBMS – shortcomings

- No logical data independence
  - Modifications to the database (schema evolution) require changes to the application and vice versa
- Lack of agreed standards, the existing standard (ODMG) is not fully implemented
- Dependence on a single programming language. Typical OODBMS is tied to a single programming language with it's programming interface
- Lack of interoperability with a large number of tools and features that are used in SQL
- Lack of Ad-Hoc queries (queries on the new tables that are obtained by joining new tables with the existing ones)

OODBMS in real world

- Chicago Stock Exchange - management in stocks trade (Versant)
- Radio Computing Services – automation of radio stations (POET)
- Ajou University Medical Canter in South Korea – all functions of the hospital, including those critical such as pathology, laboratory, blood bank, pharmacy and radiology
- CERN – big scientific data sets (Objectivity/DB)
- Federal Aviation Authority – simulation of passengers and baggage
- Electricite de France – management in electric power networks

OOSUBP products

- Versant
- Progress ObjectStore
- Objectivity/DB
- Intersystems Cachè
- POET fastObjects
- **db4o**
- Computer Associates Jasmine
- GemStone

Object-relational databases

Object-relational DBMS

- **object-relational system** (*object-relational DBMS* - ORDBMS) or *enhanced relational systems*
  - attempt to get the best of relational model and object-oriented approach
  - way of enhancing the capabilities of relational DBMS's with some of the features of object DBMS's, relation is still the fundamental abstraction
  - prototype of object-relational systems: Postgres (PostgreSQL)
  - example of extended commercial DBMS's : **Oracle, IBM** Informix

DBMS Matrix

M. Stonebreaker, D. Moore: Object-relational DBMSs – the next great wave, Morgan Kaufmann, 1996

	simple data	complex data
ad hoc query	relational database	object-relational database
no ad hoc query	file system	object oriented database

Nested relations (1)

Example of nested relation: student administration system

- each course has:
  - course ID
  - course name
  - set of lecturers
  - department

course

courseID	courseName	lecturers	dep(depID, depName)
1	NMBP	{Horvat, Hlapić}	(ZPR, Zavod za …)
2	SBP	{Horvat}	(ZPR, Zavod za …)

- domains of attributes *lecturers* and *dep* are nonatomic $\Rightarrow$ *course* relation is not in first normal form (1NF)

- 1NF version of *course*:

COURSE1NF = { courseID, courseName, personID, depID, depName }

$K_{COURSE1NF}$ = { courseID, personID}

course1NF

courseID	courseName	personID	depID	depName
1	NMBP	Horvat	ZPR	Zavod za …
1	NMBP	Hlapić	ZPR	Zavod za …
1	SBP	Horvat	ZPR	Zavod za …

- one-to-one correspondence between tuples and courses is lost
- redundancy

Nested relations (2)

- after decomposition of *course1NF*:

COURSE= { courseID, courseName, depID }
LECTURER = { courseID, personID }
DEP = { depID, depName }
PERSON= { personID }

K_{COURSE} = { courseID }
$K_{LECTURER}$ = { courseID, personID }
K_{DEP} = { depID }
K_{PERSON} = { personID}

courseN(COURSE)

courseID	courseName	depID
1	NMBP	ZPR
1	SBP	ZPR

department(DEP)

depID	depName
ZPR	Zavod za ...

person (PERSON)

personID
Horvat
Hlapić

lecturer(LECTURER)

courseID	personID
1	Horvat
1	Hlapić
2	Horvat

- to obtain information about course - joins in queries are required

Object-relational data model (1)

- Based on the relational data model
  - preserve relational foundations, such as declarative access to data
  - upward compatibility with existing relational languages
- Extend the relational data model by including object orientation and constructs to deal with added data types
  - attributes can contain complex values, including nested relation
  - increase modeling power, increase the range of applications
- no single extended relational model - all models have some concept of "object"

Object-relational data model (2)

- Extensions of relational model:
  - abstract data types (object types, structured user-defined types, ...)
  - object identity and reference types
  - methods for object types, encapsulation
  - user-defined CAST
  - object table, typed tables
  - type and table inheritance
  - nested tables (complex attributes, collection types)

- there is no DBMS's with all of extensions of relational model
  - different DBMS’s use different approaches
  - all major commercial DBMS’s implement some of extensions

SQL Standard: object-relational extensions

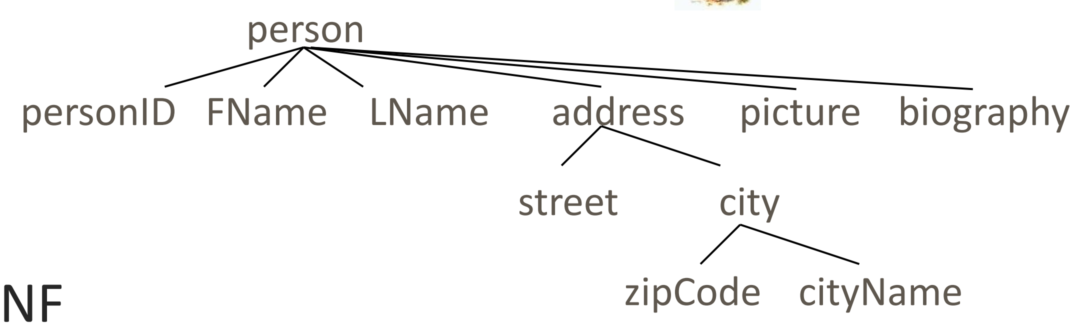
- SQL:1999 includes a number of object-oriented features
  - type constructor for specifying complex objects, object identity, encapsulation, inheritance
- every data type is either:
  - **predefined type**
    - atomic type - data type whose values are not composed of values of other data types: integer, float, character, boolean, datetime, interval ...
  - **constructed type**
    - constructed atomic type
      - *reference*
    - constructed composite type
      - collection: *array*, *multiset*
      - *row*
  - **user-defined type (UDT)**
    - *distinct type*
    - *structured type*

Example: Student administration system

- *person* relation is not in 1NF

person

personID	FName	LName	address			picture	biography
			street	city			
				zipCode	cityName		
11001	Hrvoje	Novak	Ilica 25	10000	Zagreb		Rođen je .....
78936	Ana	Kolar	Marmontova 18	21000	Split		



- person relation is not in 1NF

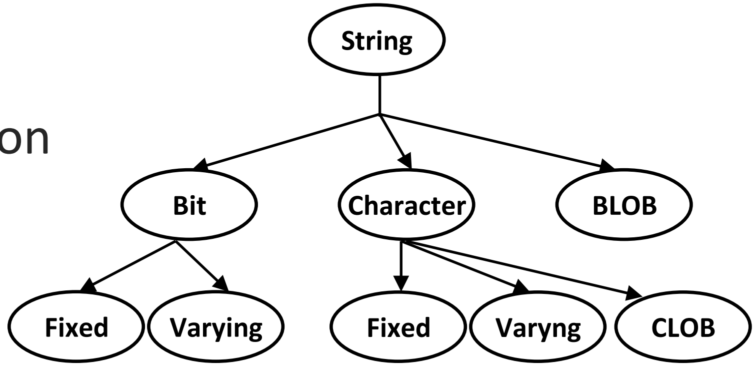
course

courseID	courseName	lectCh	lecturers	students	dep (depID, depName)
1	Napredni modeli i baze podataka	123	{123,111, 345,24}	{13503, 14111, 9000, 14678, …}	(ZPR, Zavod za prim.računarstvo)

- hierarchy of tables *person*, *lecturer* and *student*

LOB type(*Large Object Type*) (1)

- table field that holds large amount of data (text documents, graphical data such as images and computer aided designs, audio and video data)
- predefined data type (*String*)
  - ***Character Large Object* (CLOB), *National Character Large Object* (NCLOB)**
    - character strings (*printable characters*, *tabs*, *newlines*, *newpages*)
    - some standard string operation also operate on character large object strings (e.g. concatenation (||), functions SUBSTRING, UPPER, TRIM ...), comparison (LIKE, =, <>)
  - ***Binary Large Object* (BLOB)**
    - binary string that does not have a character set or collation association
    - variable length sequence of octets



LOB tip (*Large Object Type*) (2)

```
CREATE TABLE person (
        personID  INTEGER,
        FName     VARCHAR(25),
        LName     VARCHAR(25),
        biography CLOB (50K),
        picture   BLOB (2M)
);
```

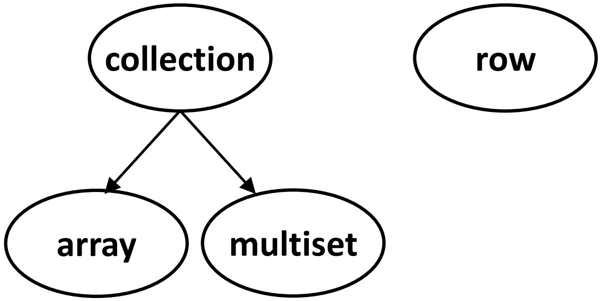
- *locator*
  - application retrieve a locator for a large object and use the locator to manipulate the object from the host language

```
EXEC SQL BEGIN DECLARE SECTION:
   SQL TYPE IS BLOB_LOCATOR
       picture_loc;
EXEC SQL END DECLARE SECTION;

EXEC SQL
   SELECT picture
     INTO :picture_loc
     FROM person
    WHERE LName = 'Horvat';

INSERT INTO person
   VALUES (..., :picture_loc);
```

- limitations
  - cannot be referenced in GROUP BY clause, ORDER BY clause, used as a part of UNIQUE constraint, foreign key, ...

SQL Standard: Constructed types

- constructed types:
  - **atomic**
    - *reference type* - **REF** *type*
      - value of REF type references (or points to) some site holding a value of the referenced type
      - the only sites that may be so referenced are the rows of typed tables (every referenced type is a structured type)
  - **composite**
    - data type whose values are composed of values than can all be of the same data type (*collection*) or of different data types (*row*)



- name of a constructed data type is a reserved word specified by standard
- specified by an type constructor (REF, ARRAY, ROW)

SQL Standard: ROW type (1)

- constructed composite type
- sequence of one or more (*field name*, *data type*) pairs, called *fields*
  - *degree* - number of elements

```
ROW ( zipCode    INTEGER,
      cityName   VARCHAR(20)
    )
```

- column of table can contain row values:

```
CREATE TABLE person (
        personID  INTEGER,
        FName     VARCHAR(25),
        LName     VARCHAR(25),
        address   ROW ( street VARCHAR(50),
                        city ROW ( zipCode      INTEGER,
                                   cityName   VARCHAR(40))
                      )
);
```

SQL Standard: ROW type (2)

- value of a ROW type consists of one value for each of its fields
- ROW constructor is used to create an instance of a ROW data type
- value of ROW element can be literal, and also result of query
- e.g.

```
ROW(10000, 'Zagreb')
```

is a value of ROW type:

```
ROW(zipCode     INTEGER,
    cityName  VARCHAR(20))
```

SQL Standard: ROW type (3)

- inserting rows into the *person* table:

```
CREATE TABLE person (
 personID INTEGER,
 FName     VARCHAR(25),
 LName     VARCHAR(25),
 address   ROW(street  VARCHAR(50),
               city ROW (zipCode    INTEGER,
                         cityName VARCHAR(40)
                         )
              )
);
```

```
INSERT INTO person VALUES(
        34562,
        'Hrvoje',
        'Novak',
         ROW('Ilica 25',
              ROW (10000,'Zagreb')
            )
);
```

- value of an element can be accessed using *dot* notation:

```
SELECT p.address.city.zipCode
  FROM person p
 WHERE p.address.street LIKE 'Ilica%';
```

SQL Standard: Collection

- constructed composite type
- comprises zero or more *elements* of a specified data type (*element type*)
- collection types:
  - ARRAY
    - one-dimensional array with a maximum number of elements
    - SQL:1999 standard
  - MULTISET
    - unordered collection that does allow duplicates
    - SQL:2003 standard
  - SET
    - unordered collection that does not allow duplicates
  - LIST
    - ordered collection that allows duplicates

SQL Standard: ARRAY (1)

- ordered collection of not necessarily distinct values, whose elements are referenced by their ordinal position in the array
- accessing elements of an array by specifying the array index
- index of element $\in$ [1, maximum cardinality]

```
CREATE TABLE course (
 courseID    INTEGER
 courseName  VARCHAR(250),
 lectCh      INTEGER REFERENCES lecturer(personID),
 lecturers   INTEGER ARRAY[10] REFERENCES lecturer(personID),
 students    INTEGER ARRAY[1000] REFERENCES student(personID),
 department  ROW(depID CHAR(8), depName VARCHAR(50)),
 PRIMARY KEY (courseID)
)
```

- two arrays of comparable element types are equal if they have the same cardinality and equal element value on every position

SQL Standard: ARRAY (2)

- inserting row with attribute of ARRAY type:

```
INSERT INTO course VALUES(1, 'Napredni modeli i baze podataka',
                          123,                   /*lecturer in charge*/
                          ARRAY[123,111,345,24], /*lecturers*/
                          ARRAY[],               /*students*/
                          ROW('ZPR','Zavod za prim.računarstvo')
);
```

```
INSERT INTO course (…, ARRAY(SELECT personID FROM lecturer WHERE …),…);
```

- updating values of elements:

```
UPDATE course
   SET students = ARRAY[13503, 14111, 9000]
WHERE courseID = 1;
```

- updating value of element at specified position:

```
UPDATE course SET students[4] = 14678
WHERE courseID = 1;
```

SQL Standard: ARRAY (3)

- CARDINALITY function - returning number of current elements

```
SELECT CARDINALITY(lecturers) AS cntLect
  FROM course
```

cntLect
4

```
UPDATE course
   SET lecturers[6] = 555
WHERE courseID = 1;

SELECT CARDINALITY(lecturers) AS cntLect
  FROM course
 WHERE courseID = 1;
```

cntLect
6

```
SELECT course.lecturers[5] AS element5
  FROM course
WHERE courseID = 1;
```

element5
NULL

- UNNEST function - converting collection to table
  - creates one row for every element of collection

SQL Standard: ARRAY (4)

- Example: UNNEST in FROM clause of SELECT statement

p.lecturers
ARRAY[123,111,345,24]

```
SELECT i.lecturerID
  FROM course AS p,
       UNNEST(p.lecturers) AS i(lecturerID)
 WHERE p.courseID = 1
```

lecturerID
123
111
345
24

```
SELECT p.courseID, i.lecturerID
  FROM course AS p,
       UNNEST(p.lecturers) AS i(lecturerID)
```

→ all (*courseID*, *lecturerID*) pairs

```
SELECT p.courseID,
       i.lecturerID, i.pozicija
  FROM course AS p,
       UNNEST(p.lecturers) WITH ORDINALITY
           AS i(lecturerID, pozicija)
```

→ all (*courseID*, *lecturerID*) pairs with position

- Example: UNNEST instead of subquery

```
SELECT courseName
  FROM course
 WHERE 123 IN (UNNEST(lecturers))
```

SQL Standard: MULTISET (1)

- unordered and unlimited collection of elements, with duplicates permitted

```
CREATE TABLE course (
 courseID     INTEGER
 courseName   VARCHAR(250),
 lectCh       INTEGER REFERENCES lecturer(personID),
 lecturers    INTEGER ARRAY[10] REFERENCES lecturer(personID),
 students     INTEGER MULTISET REFERENCES student(personID),
 department   ROW(depID CHAR(8), depName VARCHAR(50)),
 PRIMARY KEY (courseID)
)
```

  - there is no ordinal position to reference individual elements of a multiset

- two multisets of comparable element types are equal if they have the same cardinality and have the same elements, even if the elements are in different positions within the set

SQL Standard: MULTISET (2)

- inserting row with s atributom tipa MULTISET:

```
INSERT INTO course VALUES(1, 'Napredni modeli i baze podataka',
                           123,                          /*nositelj*/
                           MULTISET[123,111,345,24],     /*izvođači*/
                           MULTISET[13503, 14111, 9000], /*students*/
                           ROW('ZPR','Zavod za prim.računarstvo')
);
```

```
INSERT INTO course VALUES(1, …
                           MULTISET(SELECT personID FROM lecturer
                                     WHERE …), …);
```

- deleting last element of multiset attribute results with an empty collection (an empty collection is not equivalent to a NULL value for the column)
- some additional functions for multiset:
  - SET - removing duplicates from a multiset to produce a set
  - CARDINALITY – returning number of current elements

SQL Standard: MULTISET (3)

- Predicates for use with MULTISET:
  - comparison predicate (equality and inequality only)
  - MEMBER - checks whether a multiset contains element with specified value
    - *value1* [NOT] MEMBER [OF] *multiset_value2*
  - SUBMULTISET – checks whether one multiset is a submultiset of another
    - *multiset_value1* [NOT] SUBMULTISET [OF] *multiset_value2*
  - IS [NOT] A SET – checks whether a multiset is a set
    - *multiset_value* IS [NOT] A SET

- operators that operate on multisets:
  - MULTISET UNION [ALL | DISTINCT] – computes the union of two multisets (two variants: retain duplicates or remove duplicate)
  - MULTISET INTERSECT [ALL | DISTINCT] – intersection of two multisets
  - MULTISET EXCEPT [ALL | DISTINCT] – difference of two multisets

SQL Standard: MULTISET (4)

tabM

id	a	b	c
1	MULTISET [1,1,1,2,2,3,3]	MULTISET [1,1,2,4,4]	MULTISET [5]

```
SELECT SET (a) AS res FROM tabM
```

res
MULTISET [1,2,3]

`ELEMENT (c)`	5
`a MULTISET UNION ALL b`	MULTISET [1,1,1,1,1,2,2,2,3,3,4,4]
`a MULTISET UNION DISTINCT b`	MULTISET [1,2,3,4]
`a MULTISET INTERSECT ALL b`	MULTISET [1,1,2]
`a MULTISET INTERSECT DISTINCT b`	MULTISET [1,2]
`a MULTISET EXCEPT ALL b`	MULTISET [1,2,3,3]
`a MULTISET EXCEPT DISTINCT b`	MULTISET [1,2,3]
`b MULTISET EXCEPT ALL a`	MULTISET [4,4]

SQL Standard: MULTISET (5)

- aggregate functions for MULTISET:
  - COLLECT – for each group creates multiset from value of the argument
  - FUSION – for each group creates multiset of all multiset values in all rows
  - INTERSECTION – creates multiset intersection of a multiset value in all rows of a group

$\pi_{\text{courseID, lecturers}}(\textit{course})$

courseID	lecturers
2	MULTISET [145,245]
3	MULTISET [204,145,265]
4	MULTISET [284,145]
5	NULL

```
SELECT COLLECT(courseID) AS courses,
       FUSION(lecturers) AS fusionLect,
       INTERSECTION(lecturers) AS interLect
  FROM course
```

courses	fusionLect	interLect
MULTISET [2,3,4,5]	MULTISET [145,145,145,245,204,265,284]	MULTISET [145]

SQL Standard: Unnesting

- *unnesting* - transforming nested relation into a form with fewer (or without) complex valued attributes

- example: converting multiset into a table

```
SELECT SUM (t.c)
  FROM UNNEST (MULTISET [2, 3, 5, 7]) AS t(c)
```

→ 17

- example: unnesting of *course* table

```
SELECT courseID, courseName,
       i.lecturerID
  FROM course AS p,
       UNNEST(p.lecturers) AS i(lecturerID)
```

→ course1NF

SQL Standard: Nesting

- *nesting* - transforming 1NF relation into a nested relation

  - grouping - using COLLECT function

```
SELECT courseID, courseName,
       COLLECT(personID) AS lecturers
  FROM course1NF
GROUP BY courseID, courseName
```

→ course

COURSE1NF = { courseID, courseName, lecturer, depID, depName }

  - subquery in SELECT clause

```
SELECT courseID, courseName,
       ARRAY(SELECT personID FROM lecturer AS a
                WHERE a.courseID = k.courseID) AS lecturers
  FROM courseN AS k
```

```
or
MULTISET
```

LECTURER = { courseID, personID }

SQL Standard: User-defined types

- User-defined types (UDT)
  - the name of a user-defined type is provided in its definition
  - persistent
  - sometimes called *abstract data types*

(1) **distinct type**
- based on a single predefined data type called *source type*
- type inheritance is not allowed

(2) **structured type**
- expressed as a list of attributes
- type inheritance is allowed

SQL Standard: Creating of user-defined types

```
CREATE TYPE <user-defined type body>

<user-defined type body> ::= <user-defined type name> [UNDER <supertype name>]
                             [AS <representation>] [[NOT] INSTANTIABLE]
                             [NOT] FINAL
                             [REF IS SYSTEM GENERATED | REF USING <predefined type> |
                                  REF FROM <attribute name> [{,<attribute name> }...]]
                             [<method specification list>]

<representation> ::= <predefined type> | <member list>

<member list> ::= (<attribute definition> [{,<attribute definition> }...])

<attribute definition> ::= <attribute name> {<data type> | <collection type>}
                           [<reference scope check>] [DEFAULT <default value>]

<data type> ::= <predefined type> | <reference type>

<collection type> ::= <data type> ARRAY [<unsigned integer>]   /* [] dio sintakse */

<method specification list> ::= <method specification> [{,<method specification> }... ]

<method specification> ::=
    <partial method specification> | <overriding method specification>

<overriding method specification> ::= OVERRIDING <partial method specification>

<partial method specification> ::= [CONSTRUCTOR] METHOD <method name>
                                    <SQL parameter declarations> RETURNS <data type>
```

SQL Standard: DISTINCT type (1)

- user-defined type - more limited variant
- based on some predefined type (*source type*)
- renamed type, with different behavior than its source type
  - allows differentiation between the same underlying base types
- methods can be defined over DISTINCT types
- type inheritance is not allowed

```
CREATE TYPE ageT    AS INTEGER FINAL;
CREATE TYPE weightT AS INTEGER FINAL;
CREATE TABLE person (
        personID     INTEGER
        LName        VARCHAR(25),
        personAge    ageT
        personWeight weightT
        PRIMARY KEY personID);
```

SQL Standard: DISTINCT type (2)

- comparison of the values of the same DISTINCT type:

```
SELECT p1.personID
  FROM person p1, person p2
 WHERE p2.personID = 123 AND p1.personAge < p2.personAge;
```

- values cannot be directly mixed in operations with source type or other distinct types;

```
SELECT personID FROM person
 WHERE (personAge * 2) < personWeight;
```

→ ERROR

  - using CAST function is needed:

```
SELECT personID FROM person
 WHERE CAST(personAge AS INTEGER) * 2 < CAST(personWeight AS INTEGER);
```

SQL Standard: Structured types (1)

- named, user-defined data type
- composed of one or more *attributes*
  - attribute has name and data type
    - example: definition of *cityT* structured type

```
CREATE TYPE cityT AS (
        zipCode     INTEGER,
        cityName  VARCHAR(40))
  NOT FINAL;
```

- **NOT FINAL** - creating of subtypes of *cityT* type is allowed

- instances of structured type are values, with some characteristics of objects
  - stored data ⇒ state ⇒ attributes
  - behavior ⇒ semantics ⇒ methods
- value of a structured type comprises a number of attribute values

SQL Standard: Structured types (2)

- structured type can be used as:
  - type of attributes of other structured types
  - type of parameters of functions, methods, and procedures
  - type of SQL variables
  - type of columns in tables and type of table rows
- example - structured type as column type:

```
CREATE TYPE cityT AS (
        zipCode   INTEGER,
        cityName  VARCHAR(40)
) NOT FINAL;

CREATE TYPE addressT AS (
        street    VARCHAR(50),
        city      cityT
) NOT FINAL;
```

```
CREATE TABLE person (
        personID  INTEGER
        FName     VARCHAR(25),
        LName     VARCHAR(25),
        address   addressT
);
```

SQL standard: Typed tables (1)

- ***typed table*** - table that is declared to be based on some structured type
  - contains columns that correspond to the associated type's attributes
  - additional column - *self-referencing column* - value that uniquely identifies the row in which it is stored
    - rows in different typed tables may have equal values in their self-referencing column

- creating typed tables - SQL sintax:

```
CREATE TABLE <table name> OF <user-defined type>
   [UNDER <supertable name>][ <table element list>]
<table element list> ::= (<table element> [{,<table element>}...])
<table element> ::= <table constraint>
   | <self-referencing column specification> | <column options>
<self-referencing column specification > ::=
   REF IS <self-referencing column name> <reference generation>
<reference generation> ::= SYSTEM GENERATED | USER GENERATED | DERIVED
<column options> ::= <column name> WITH OPTIONS <column option list>
<column option list> ::= [SCOPE <table name>[<reference scope check>]]
                [DEFAULT <default value>] [ <column constraint>... ]
```

SQL standard: Typed tables (2)

- Example: creating *city* table based on *cityT* type

1.
```
CREATE TYPE cityT AS (zipCode     INTEGER,
                      cityName  VARCHAR(40))
                      INSTANTIABLE
                      NOT FINAL
                      REF IS SYSTEM GENERATED;
```

  - INSTANTIABLE – instances of the type can be created (system-supplied *constructor* method is generated);
  - NOT INSTANTIABLE - no system-supplied *constructor* method is generated

- three different mechanisms by which unique identities are given to instances of the structured types - REF IS clause:
  - SYSTEM GENERATED - automatically generated by system
  - USING - user generated
  - FROM - derived from one or more attributes of the structured type

SQL standard: Typed tables (3)

2.

```
CREATE TABLE city OF cityT
(PRIMARY KEY (zipCode),
 REF IS cityOID SYSTEM GENERATED,
 zipCode WITH OPTIONS CONSTRAINT permZipCode
                    CHECK (zipCode BETWEEN 10000 AND 99999));
```

- OF clause - specifies the structured type whose instances can be stored in the table
- REF IS clause - for specifying name and type of the self-referential attribute (must be consistent with the mechanism specified in definition of used structured type)
  - SYSTEM GENERATED, USER GENERATED, DERIVED

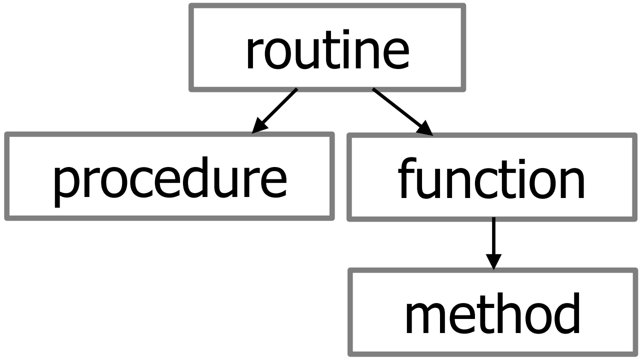
- inserting row into *city* table:

```
INSERT INTO city VALUES (10000, 'Zagreb');
```

city

cityOID	zipCode	cityName
1023456734	10000	Zagreb

SQL-invoked routines (1)



- SQL-invoked routines - routine that is invoked from SQL code
- three principle classes of SQL-invoked routine:
  - **Function:**
    - only input parameters (return a single value as the "value" of a function invocation)
    - can be called using function notation: **`functionName (parameters)`**
  - **Method:**
    - special sort of function - function that is closely associated with a single structured type
    - every method always has one implicit parameter, whose data type must be the associated type
  - **Procedure:**
    - input and output parameters
    - typically invoked using some form of CALL statement

SQL-invoked routines (2)

- SQL-invoked routines can be written in:
  - SQL - **SQL routines**
    - database systems support their own procedural languages (e.g. Oracle -PL/SQL, Microsoft SQLServer - TransactSQL, u IBM Informix - SPL)
  - other programming language (e.g. Java, C) - **external routines**
    - using routines already written and created for a different purpose, or perform computationally intensive tasks more efficiently than SQL routines
    - more portable among SQL database systems from different vendors
    - problem of mismatching of data types (e.g. time, blob, …)
    - external routine is defined *by specifying an EXTERNAL clause*
    - *example* - specifying of external routine:

```
CREATE FUNCTION smallPicture (IN picture pictureT) RETURNS BOOLEAN
EXTERNAL NAME '/usr/bin/pictures/smallPic'
LANGUAGE C ...
```

SQL standard: Structured types - Methods

- implementation of a structured type is hidden from the user
  - structured types are manipulated by invoking methods defined on them
    - type of the first, implicit parameter of the method is associated structured type (functions and procedures are not associated with structured type)
  - change of implementation does not affect the application if the interface remains unchanged
  - attribute values are *encapsulated* - attribute value is accessible only by invoking *observer* function and *mutator* function

SQL standard: Implicit methods (1)

- three types of implicit methods - automatically generated when type is defined
  - ***constructor***
    - creates instance of the type
    - same name as the data type
    - method with no arguments
    - sets the attributes to their default values (or NULL values)
    - invoked using the *new* keyword
  - ***observer*** and ***mutator***
    - one for each attribute of the structured type
    - same name as its associated attribute
    - *observer* - retrieve value of attribute
    - *mutator* - change value stored in attribute
    - invoked using *dot* notation (**variable.functionName**)

SQL standard: Implicit methods (2)

- Example:
  - using of sequence of SQL routine with calling *construcor* method and *mutator* method for creating instance of the type *cityT*

```
CREATE TYPE cityT AS (
        zipCode     INTEGER,
        cityName  VARCHAR(40)
) NOT FINAL;
CREATE TABLE person (
        personID   INTEGER
        FName      VARCHAR(25),
        LName      VARCHAR(25),
        cityPerson cityT
);
```

```
BEGIN
  DECLARE iMjesto cityT;
  /* generation of a new instance */
  SET iMjesto = new cityT();
  /* attrbutes are modified by invoking
     mutator methods */
  iMjesto.zipCode(10000);
  iMjesto.cityName('Zagreb');
  ...
  INSERT INTO person VALUES(…, iMjesto);
END
```

  - constructor creates a value of the type, not an object of the type
    - objects correspond to rows of a *typed table*

SQL standard: User-defined methods (1)

- structured type can have user-defined methods defined on it
- method is declared as part of the definition of its associated structured type

```
CREATE TYPE employeeT (FName   VARCHAR(15),
                       LName   VARCHAR(15),
                       salary  INTEGER) NOT FINAL
   METHOD newSalary(percent INTEGER) RETURNS INTEGER;
```

- CREATE METHOD statement - creating method body:

```
CREATE METHOD newSalary (percent INTEGER) RETURNS INTEGER
       FOR employeeT
  BEGIN
     RETURN (100 + percent) * SELF.salary / 100;
  END
```

  - keyword SELF refers to the structured type instance on which the method is invoked

- method invocation: retriving salary with 7% raise for each employee

```
{ CREATE TABLE employee OF employeeT; }
SELECT LName, newSalary (7)
  FROM employee
```

SQL standard: User-defined methods (2)

- Example: overriding the constructor method
  - setting of attribute values at the time of creating an type instance
  - keyword OVERRIDING must be specified in type definition

```
CREATE TYPE cityT AS (
        zipCode     INTEGER,
        cityName  VARCHAR(40)) NOT FINAL
OVERRIDING constructor METHOD /* new constructor with parameters */
   cityT(zipCode INTEGER, cityName VARCHAR(40)) RETURNS cityT;
```

```
CREATE METHOD cityT (pZipCode INTEGER, pCityName VARCHAR(40))
       RETURNS cityT FOR cityT
  BEGIN
     SET SELF.zipCode = pZipCode;
     SET SELF.cityName = pCityName
     RETURN SELF;   /* modified instance of cityT type */
  END
```

```
...
DECLARE newCity cityT;
SET newCity = new cityT (10000, 'Zagreb');
...
```

SQL standard: Using methods

- Example: using *constructor* and *mutator* methods

```
CREATE TYPE cityT AS (
      zipCode     INTEGER,
      cityName  VARCHAR(40)
) NOT FINAL;
CREATE TABLE person (
      personID   INTEGER
      FName      VARCHAR(25),
      LName      VARCHAR(25),
      cityPerson cityT
);
```

```
INSERT INTO person
     VALUES (12345, 'Pero', 'Perić'
                 NEW cityT(10000, 'Zagreb'))
```

```
UPDATE person SET cityPerson
      = NEW cityT(10010, cityPerson.cityName)
 WHERE personID = 12345
```

```
...SET cityPerson = cityPerson.zipCode(10010) ...
```

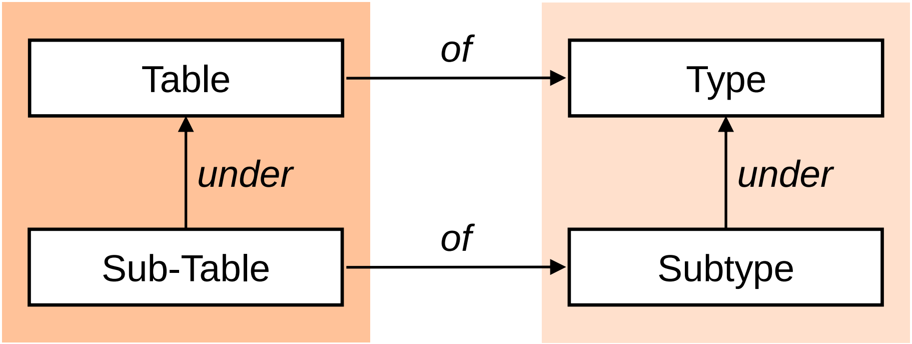
```
...SET cityPerson.zipCode = 10010 ...
```

- using *observer* method

```
SELECT p.cityPerson.zipCode
  FROM person p
 WHERE p.cityPerson.cityName LIKE '%Zagreb%';
```

SQL standard: Inheritance

- type inheritance
  - allowed only for structured types
  - type hierarchy
- table inheritance
  - allowed only for typed tables
  - table hierarchy
  - analogy with specialization/generalization in the E-R model
  - allows multiple types of the same object and simultaneous existence of the same entity in more than one table



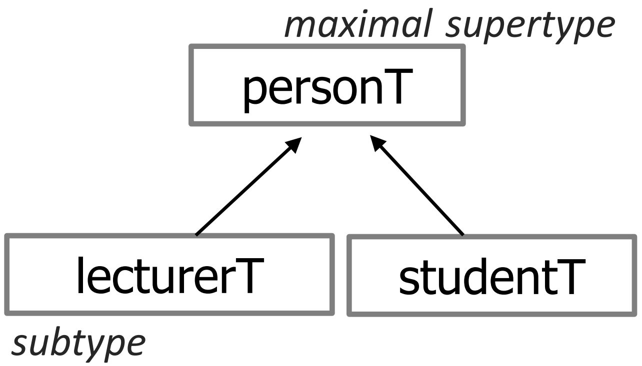
SQL standard: Type inheritance (1)

- subtype inherit structure (attributes) and behavior (methods) from their supertype
- inheritance is specified via the UNDER keyword in the structured type definition
- subtype of some type can be created under condition that NOT FINAL keyword is specified in type definition
- subtype can redefine effect of methods by redeclaration, using method overriding
  - in the definition of subtypes, instead METHOD, OVERRIDING METHOD is used

Type inheritance (2)

- Example:
  - person can be student or lecturer (student and lecturer have some common and some additional different features)
  - defining type hierarchy :

```
CREATE TYPE personT AS (
        personID   INTEGER,
        jmbg       CHAR(13)
        LName      VARCHAR(25))
        INSTANTIABLE NOT FINAL
        REF IS SYSTEM GENERATED
METHOD age(jmbg CHAR(13)) RETURNS INTEGER;

CREATE TYPE studentT UNDER personT AS (
        studyProgram CHAR(20),
        department   VARCHAR(100))
        INSTANTIABLE NOT FINAL;

CREATE TYPE lecturerT UNDER personT AS (
        salary      INTEGER,
        department  VARCHAR(100))
        INSTANTIABLE NOT FINAL;
```



- *studentT* and lecturer*T* are subtypes of the *personT type*
- *personT* is supertype of *studentT* and lecturer*T*
- *studentT* and lecturer*T* inherits from *personT* :
  - attributes *personID, jmbg* and *LName*
  - implicit methods, method *age*

SQL standard: Table inheritance

- allowed only for typed tables
- structured type of the direct supertable of some typed table must be the direct supertype of the structured type associated with that subtable
- subtable inherits the columns of its parent tables
- every row in a subtable is also a row in each of its parent tables
  - tuples in a subtable corresponds to tuples in a parent table if they have the same values for all inherited attributes
  - corresponding tuples represent the same entity

Example: creating *student* and *lecturer* tables as subtables of *person* table

```
CREATE TABLE person OF personT
(PRIMARY KEY (personID),
 REF IS personOID SYSTEM GENERATED;

CREATE TABLE student OF studentT
             UNDER person;

CREATE TABLE lecturer OF lecturerT
             UNDER person;
```

- first must be defined type hierarchy of *personT, studentT and lecturerT* types
- attributes *personID*, *jmbg* and *LName* exists also in *student* and *lecturer* tables
- every row in *student* and *lecturer* is also a row in *person* table

SQL standard: Inheritance - some rules

- multiple inheritance is not allowed - subtype/subtable inherits from one supertype/supertable
- primary key is defined only for the maximal supertable, and inherited by all of its subtables
- REF IS clause is defined only for the maximal supertype/supertable. Object identifier is inherited by all subtables
- in table definition, integrity constraints can be specified only for originally defined columns (not for inherited columns)
- type hierarchy can be defined independently of the table hierarchy
  - NOT INSTANTIABLE clause can be used in the hierarchy of types which are not related to typed tables
- all types associated with the hierarchy of typed tables must be defined as INSTANTIABLE

SQL standard: Table hierarchy - inserting rows

- INSERT statement must include required values for the original and the inherited attributes
  - if REF IS SYSTEM GENERATED, value of the object identifier is automatically generated during the INSERT operation
- *most-specific table* of row - table in which that row is directly inserted
- *most-specific type* of row - each value must be associated with one specific type, corresponds to the lowest sub-type assigned to the instance

```
INSERT INTO person VALUES (111, '120196', 'Kolar');
INSERT INTO person VALUES (222, '231196', 'Novak');
INSERT INTO lecturer VALUES (555, '300176, 'Jurak', 5000, 'ZPM');
```

person

personOID	personID	jmbg	LName
123456	111	120196	Kolar
234567	222	231196	Novak
564790	555	300176	Jurak

lecturer

personOID	personID	jmbg	LName	salary	department
564790	555	300176	Jurak	5000	ZPM

SQL standard: Table hierarchy - retrieving rows

- query on a supertable can include attributes (original and inherited) only from this supertable (not from subtables)
- result of the query on a supertable may include:
  - rows from the supertable and its subtables, or
  - rows only from supertable (i.e. rows with no subrows in subtables)
    - keyword **ONLY** in FROM clause of SELECT statement

```
SELECT jmbg
     , LName
  FROM person
```

jmbg	LName
111	Kolar
222	Novak
555	Jurak

```
SELECT jmbg
     , LName
  FROM lecturer
```

jmbg	LName
555	Jurak

- retrieving a person who is neither a student nor teacher:

```
SELECT jmbg
     , LName
  FROM ONLY(person)
```

jmbg	LName
111	Kolar
222	Novak

SQL standard: REF type (1)

- value of REF type references (or points to) some site holding a value of the referenced type
  - if *T* is a type, then REF *T* is the type of a reference to *T*, that is, a pointer to an object of type *T*
- pointers only to rows in typed tables
- SQL syntax for specifying a REF type:

```
<reference type> ::= REF (<referenced type>) [ <scope clause> ]
           [ARRAY [<unsigned integer>]] /* [] je dio sintakse */
           [ <reference scope check> ]
<referenced type> ::= <user-defined type name>
<scope clause> ::= SCOPE <table name>
<reference scope check> ::= REFERENCES ARE [ NOT ] CHECKED
                            [ ON DELETE <action> ]
```

- data type of self referencing columns is always a REF type
- can be used as type of:
  - columns in ordinary SQL tables
  - attributes of structured types
  - SQL variables, parameters

SQL standard: REF type (2)

```
CREATE TYPE courseT (
   courseID      INTEGER,
   courseName    VARCHAR(250),
   lectCh        REF (lecturerT),
   lecturers     REF (lecturerT) ARRAY[10],
   students      REF (studentT) MULTISET,
   department    departmentT)
   INSTANTIABLE NOT FINAL
   REF IS SYSTEM GENERATED
```

Note: *departmentT* structured type and table hierarchy of *person, lecturer* and *student typed table* are already created

```
CREATE TABLE course OF courseT
(PRIMARY KEY (courseID)
 REF IS courseOID SYSTEM GENERATED)
```

→ using of REF type to define a relationship with objects of *lecturerT* type, and objects of *studentT* type

course

courseOID	courseID	courseName	lectCh	...
1463144246	1	Napredni ...		

to a *lecturerT* object

SQL standard: REF type (3)

- inserting row with NULL value for attribute defined as REF type:

```
INSERT INTO COURSE (courseID, courseName, lectCh)
       VALUES (1, 'Napredni modeli i baze podataka', NULL);
```

- updating value of attribute defined as REF type:

```
UPDATE course
   SET lectCh = (SELECT personOID
                FROM lecturer
               WHERE personID = 12343)
 WHERE courseName = 'Napredni modeli i baze podataka'
```

  - using query we can get the identifier value of a row

```
CREATE TABLE person OF personT (PRIMARY KEY (personID),
       REF IS personOID SYSTEM GENERATED;
CREATE TABLE student OF studentT UNDER person;
CREATE TABLE lecturer OF lecturerT UNDER person;
```

SQL standard: REF type (4)

- scope of REF type:
  - associated structured type of the REF type's referenced table must be the REF type's specified referenced type
  - SCOPE clause in specification of REF type in CREATE TYPE and CREATE TABLE statements - specifies the name of a typed table whose associated structured type is the referenced type of the REF type
  - REFERENCES ARE CHECKED clause - system examines the referenced table to ensure that there is a row in that table whose self-referencing column value is equal to the reference value
    - incorrect values of references are not allowed
  - ON DELETE clause - action to be invoked when the row identified by a reference value is delete:
    - NO ACTION, SET NULL, SET DEFAULT, CASCADE

SQL standard: REF type (5)

- example: scope of REF type

  - CREATE TYPE statement

```
CREATE TYPE courseT (
 ...
 lectCh REF (lecturerT) SCOPE lecturer
                           REFERENCES ARE CHECKED
                           ON DELETE SET NULL
 ...)
```

  - CREATE TABLE statement:

```
CREATE TABLE course OF courseT
                  (lectCh WITH OPTIONS SCOPE lecturer
                                 REFERENCES ARE CHECKED
                                 ON DELETE SET NULL)
```

SQL standard: REF type (6)

```
CREATE TYPE courseT (... lectCh REF (lecturerT) ...);
CREATE TABLE course OF courseT ...
```

- retrieval using reference value
  - users are interested in attribute values of referenced row
  - → (*dereference operator*) is using to access the columns of the referenced table
    - implicit join - query retrieves a value from referenced table without specifying that table in FROM clause statement

```
SELECT lectCh -> LName, lectCh -> newSalary(7)
  FROM course
 WHERE courseName = 'Napredni modeli i baze podataka';
```

method from *lecturerT* type

  - *deref* function - takes a reference as an argument and returns a row of the type pointed to

```
SELECT deref(lectCh)
  FROM course p
 WHERE p.department.depID = 'ZPR';
```

→ query result is table with one column of type *lecturerT*

ORDBMS advantages and disadvantages

- advantages
  - main advantages come from *reuse* and *sharing*
    - functionality embedded in the server can be shared by all applications
  - all possibilities of relational databases preserved
    - the extended relational approach preserves the significant body of knowledge and experience that has gone into developing relational applications
- disadvantages
  - complexity
  - dissatisfaction of relational model supporters
    - basic simplicity and purity of relational model is lost
    - lower performance than the current relational technology
  - dissatisfaction of object-oriented model supporters
    - dissatisfaction with used terminology and object concepts

Literature

- S.W. Dietrich, S.D. Urban: **An Advanced Course in Database Systems : Beyond Relational Databases**, Prentice Hall, 2005
- Jim Melton: **Advanced SQL: 1999 - Understanding Object-Relational and Other Advanced Features**, Morgan Kaufmann, 2002
- T. Connolly, C. Begg: **Database Systems: A Practical Approach to Design, Implementation, and Management**, 4th Edition, Pearson Education , 2005
- A. Silberschatz, H.F. Korth, S. Sudarshan: **Database Systems Concepts**, 5th Edition, McGraw-Hill, 2005.
- M. Stonebraker, D. Moore: **Object-Relational DBMSs: The Next Great Wave**, Morgan Kaufmann Publichers, 1996
- R. Ramakrishnan, J. Gehrke: **Database Management Systems**, McGraw-Hill, 2003
- **PostgreSQL 9.3.5 Documentation -** http://www.postgresql.org/docs/9.3/static/index.html